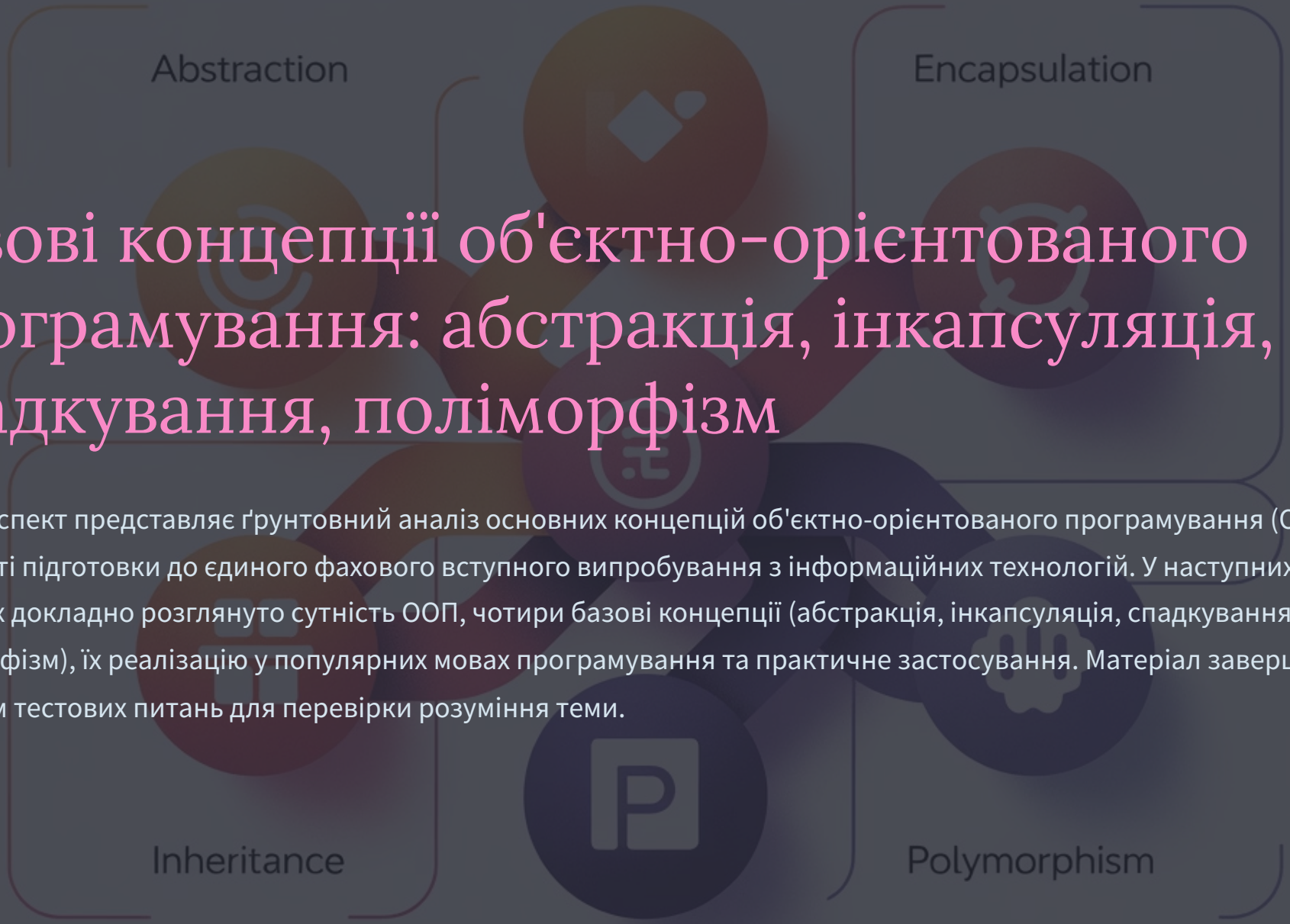


Базові концепції об'єктно-орієнтованого програмування: абстракція, інкапсуляція, спадкування, поліморфізм

Цей конспект представляє ґрунтовний аналіз основних концепцій об'єктно-орієнтованого програмування (ООП) у контексті підготовки до єдиного фахового вступного випробування з інформаційних технологій. У наступних розділах докладно розглянуто сутність ООП, чотири базові концепції (абстракція, інкапсуляція, спадкування, поліморфізм), їх реалізацію у популярних мовах програмування та практичне застосування. Матеріал завершується набором тестових питань для перевірки розуміння теми.

Вступ до об'єктно-орієнтованого програмування (ООП)

Об'єктно-орієнтоване програмування (ООП) виникло як відповідь на зростаючу складність програмного забезпечення у 1960-х роках. Перша мова програмування, що включала концепції ООП, Simula, була створена норвезькими вченими Оле-Йоханом Далем і Крістеном Нюгаардом. Проте справжній розквіт ООП відбувся з появою мови Smalltalk у 1970-х, розробленої Аланом Кеєм та його командою в Xerox PARC.

Еволюція ООП продовжилась із появою C++ у 1980-х, створеною Б'ярном Страуструпом як розширення мови C. Наступним важливим кроком стала поява Java у 1995 році, розробленої Джеймсом Гослінгом у Sun Microsystems, що додатково популяризувала ООП. Сьогодні ООП інтегровано у більшість сучасних мов програмування, включаючи Python, C#, Ruby, та інші.

ООП стало важливим у сучасному програмуванні з кількох причин. По-перше, воно дозволяє створювати модульні, повторно використовувані компоненти коду, що значно полегшує розробку складних систем. По-друге, ООП сприяє більш інтуїтивному моделюванню реального світу в коді, де об'єкти мають властивості та поведінку. По-третє, воно забезпечує кращу організацію коду, що полегшує його підтримку та розширення.

Основними рисами мов ООП є підтримка класів і об'єктів, абстракції, інкапсуляції, спадкування та поліморфізму. Ці концепції дозволяють програмістам створювати гнучкі, масштабовані та легко підтримувані програмні системи. Саме тому розуміння принципів ООП є фундаментальним для будь-якого програміста, незалежно від мови програмування, яку він використовує.

Основи ООП у контексті мов програмування

Мови програмування є формальними системами, що задають набір правил і синтаксис для створення програм, які можуть бути виконані комп'ютером. Залежно від підходу до вирішення задач, мови програмування можна класифікувати за різними парадигмами: процедурні, функціональні, логічні та об'єктно-орієнтовані. Останні фокусуються на концепції об'єктів — сутностей, що поєднують стан (дані) та поведінку (методи).

Java

Чиста об'єктно-орієнтована мова з суворим дотриманням принципів ООП. Використовується для розробки корпоративних додатків, мобільних додатків (Android), веб-серверів та інших систем.

C++

Гібридна мова, що підтримує як процедурне, так і об'єктно-орієнтоване програмування. Широко використовується для системного програмування, розробки ігор та програм, що вимагають високої продуктивності.

Python

Мультипарадигмальна мова, що підтримує ООП, але також функціональний та процедурний стилі. Популярна для веб-розробки, аналізу даних, машинного навчання та автоматизації.

C#

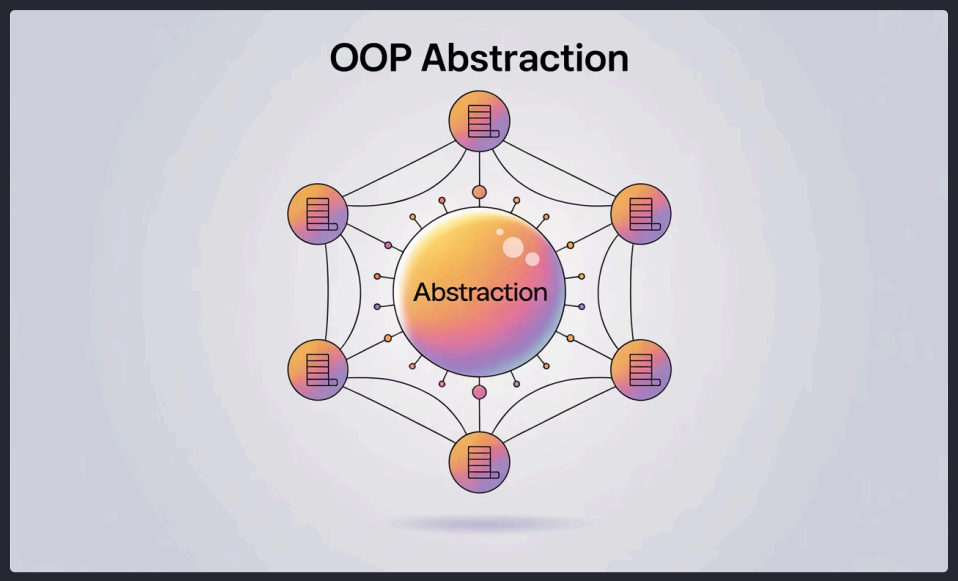
Сучасна мова від Microsoft, спроектована як об'єктно-орієнтована з елементами функціонального програмування. Використовується для розробки Windows-додатків, веб-додатків ASP.NET та ігор на Unity.

ООП як парадигма програмування пропонує особливий погляд на структуру програми. Замість того, щоб розглядати програму як послідовність інструкцій (як у процедурному програмуванні) чи набір функцій (як у функціональному), ООП бачить програму як колекцію взаємодіючих об'єктів. Ці об'єкти є екземплярами класів — шаблонів, що визначають структуру та поведінку категорій об'єктів.

Основна філософія ООП полягає в моделюванні програми таким чином, щоб вона відображала реальні чи концептуальні сутності та їх взаємодії. Це досягається завдяки чотирьом фундаментальним концепціям: абстракція, інкапсуляція, спадкування та поліморфізм. Ці принципи дозволяють створювати модульний, повторно використовуваний код, що легко адаптується до змін вимог. У наступних розділах ми детально розглянемо кожну з цих концепцій.

Абстракція: сутність та призначення

Абстракція — це фундаментальна концепція ООП, яка дозволяє моделювати складні системи шляхом виділення найважливіших аспектів об'єкта і ігнорування несуттєвих деталей. Вона дає можливість представити складну реальність у вигляді спрощених моделей, з якими легше працювати у програмуванні. Абстракція зосереджується на тому, що об'єкт робить, а не як він це робить.



У контексті ООП абстракція реалізується через класи та інтерфейси. Клас — це абстрактна модель сутності, що визначає її властивості (атрибути) та поведінку (методи). Інтерфейс, у свою чергу, визначає лише поведінку без конкретної реалізації, створюючи своєрідний "контракт", який мають виконувати конкретні класи.

Наприклад, розглянемо абстракцію "Транспортний засіб". Вона може мати такі властивості як швидкість, вага, розмір, та такі методи як "рухатися", "зупинятися", "повертати". Конкретні реалізації (автомобіль, велосипед, літак) матимуть свої специфічні властивості та деталі реалізації методів, але всі вони відповідатимуть загальній абстракції транспортного засобу.

Спрощення складних систем

Абстракція дозволяє зосередитись на ключових аспектах системи, ігноруючи несуттєві деталі, що робить дизайн програми більш зрозумілим і керованим.

Керування складністю

Абстракція дозволяє розбити складну систему на окремі, добре визначені компоненти, що полегшує її розуміння та розробку.

Повторне використання коду

Правильно спроектовані абстракції можуть бути повторно використані в різних частинах програми або навіть у різних проектах.

Створення ієрархій

Абстракція сприяє побудові логічних ієрархій концепцій, що полегшує організацію коду та відображає природні відносини між сутностями.

Важливо розуміти, що абстракція — це не лише технічний механізм, але й спосіб мислення про програмні системи. Вона дозволяє програмістам створювати зрозумілі та гнучкі моделі, що можуть еволюціонувати разом із розвитком програмного забезпечення. Правильно спроектовані абстракції є ключем до створення якісного, масштабованого програмного забезпечення.

Реалізація абстракції у популярних мовах

У різних мовах програмування механізми реалізації абстракції мають свої особливості, проте загальні принципи залишаються схожими. Розглянемо, як створюються абстрактні класи та інтерфейси у найпопулярніших об'єктно-орієнтованих мовах.

Java

У Java абстрактні класи створюються за допомогою ключового слова **abstract**. Такі класи можуть містити як абстрактні методи (без реалізації), так і звичайні методи з повною реалізацією. Інтерфейси створюються за допомогою ключового слова **interface** і традиційно містили лише абстрактні методи без реалізації, хоча з Java 8 можуть включати методи за замовчуванням.

C++

У C++ абстрактні класи створюються шляхом оголошення хоча б одного чистого віртуального методу з використанням синтаксису **virtual void method() = 0;**. Немає окремого типу "інтерфейс", але абстрактний клас без даних і з усіма методами чисто віртуальними функціонально еквівалентний інтерфейсу.

Python

Python з версії 3.4 підтримує абстрактні базові класи через модуль **abc**. Клас стає абстрактним, якщо він успадковується від **ABC** і містить методи, декоровані **@abstractmethod**. У Python немає формального поняття інтерфейсу, але абстрактні базові класи часто використовуються для цієї мети.

C#

У C# абстрактні класи створюються за допомогою ключового слова **abstract**. Вони можуть містити абстрактні методи (без реалізації) та звичайні методи. Інтерфейси створюються за допомогою ключового слова **interface** і традиційно містили лише сигнатури методів без реалізації, хоча з C# 8.0 можуть мати методи за замовчуванням.

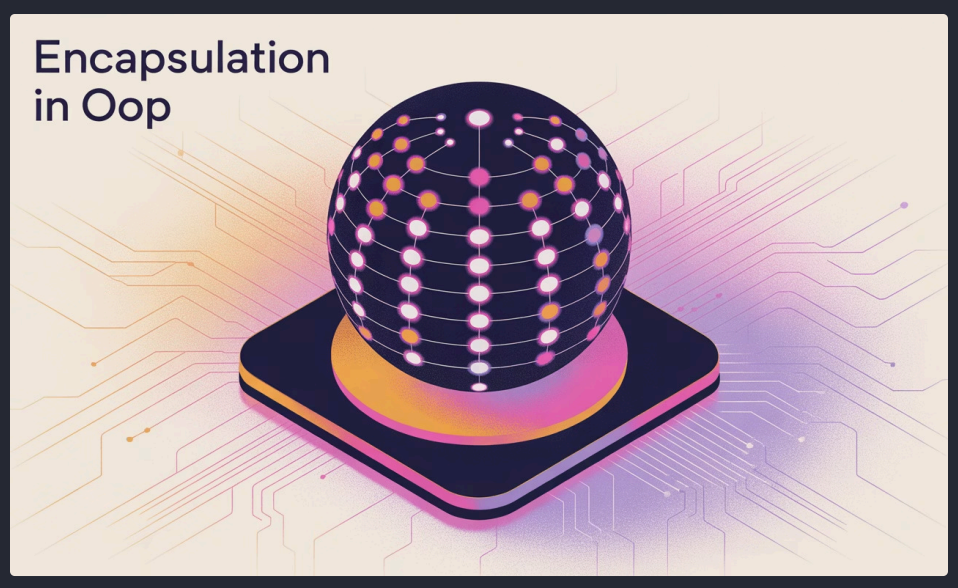
Приклад абстрактного класу в Java:

```
// Абстрактний клас "Транспортний засіб"
public abstract class TransportVehicle {
    // Звичайні властивості
    protected int speed;
    protected double weight;

    // Звичайний метод з реалізацією
    public void setSpeed(int speed) {
        this.speed = speed;
    }

    // Абстрактний метод без реалізації
    public abstract void move();

    // Абстрактний метод без реалізації
    public abstract void stop();
}

// Конкретний клас, що успадковує абстрактний
public class Car extends TransportVehicle {
    private int numberOfDoors;

    // Реалізація абстрактного методу
    @Override
    public void move() {
        System.out.println("Автомобіль рухається на колесах");
    }

    // Реалізація абстрактного методу
    @Override
    public void stop() {
        System.out.println("Автомобіль зупиняється за допомогою гальм");
    }
}
```

Абстракція в ООП є потужним інструментом проектування, що дозволяє створювати гнучкі, розширювані системи. Правильне використання абстрактних класів та інтерфейсів сприяє кращій організації коду та полегшує його підтримку й розширення в майбутньому.

Інкапсуляція: захист та організація даних

Інкапсуляція — це один з фундаментальних принципів ООП, який полягає в об'єднанні даних (атрибутів) і методів, що працюють з цими даними, в єдиний об'єкт, і обмеженні прямого доступу до внутрішніх даних об'єкта ззовні. Простіше кажучи, інкапсуляція дозволяє "сховати" внутрішню реалізацію об'єкта, надаючи доступ до функціональності лише через чітко визначений інтерфейс.



Основна мета інкапсуляції — забезпечити контроль над даними, запобігаючи їх неправильному використанню та модифікації. Це досягається через механізми контролю доступу, які визначають, які частини коду можуть взаємодіяти з конкретними властивостями та методами.

Private (Приватний)

Приватні члени класу доступні лише всередині самого класу. Вони повністю скриті від зовнішнього коду та від підкласів. Це найсильніший рівень інкапсуляції, що забезпечує повний контроль над внутрішнім станом об'єкта.

Protected (Захищений)

Захищені члени класу доступні всередині класу та в усіх його підкласах. Вони недоступні для зовнішнього коду. Цей рівень доступу дозволяє підкласам використовувати або модифікувати певні аспекти батьківського класу.

Public (Публічний)

Публічні члени класу доступні з будь-якого місця програми. Вони формують публічний інтерфейс класу — те, як інші частини програми взаємодіють з об'єктами цього класу.

Package/Default (Пакетний)

У деяких мовах (Java, C#) є додатковий рівень доступу, який дозволяє доступ до членів класу лише з того ж пакету або простору імен. Це корисно для організації коду в логічні групи.

Переваги інкапсуляції для розробки програмного забезпечення важко переоцінити:

- **Захист даних від випадкових змін.** Контролюючи доступ до даних через методи, можна запобігти помилковим модифікаціям, які могли б порушити внутрішню логіку об'єкта.
- **Гнучкість та простота обслуговування.** Внутрішня реалізація класу може бути змінена без впливу на код, що використовує цей клас, якщо публічний інтерфейс залишається незмінним.
- **Контроль над станом об'єкта.** Методи доступу можуть включати валідацію, щоб гарантувати, що об'єкт завжди знаходиться у коректному стані.
- **Приховування складності.** Складні внутрішні механізми можуть бути приховані за простим інтерфейсом, що спрощує використання класу.
- **Підтримка зміни внутрішньої реалізації.** Зовнішній код залежить лише від публічного інтерфейсу, тому внутрішня реалізація може еволюціонувати без порушення існуючого коду.

Інкапсуляція є ключовим принципом для створення надійного, масштабованого та легко підтримуваного коду. Вона дозволяє розробникам абстрагуватися від внутрішніх деталей реалізації та зосередитись на використанні об'єктів через їх чітко визначені інтерфейси.

Методи реалізації інкапсуляції

Основним механізмом реалізації інкапсуляції є використання модифікаторів доступу та спеціальних методів для контрольованого доступу до внутрішніх даних об'єкта. Розглянемо детальніше ці методи та їх реалізацію у різних мовах програмування.

Геттери (Accessors)

Методи, що дозволяють отримати значення приватних полів. Вони надають контрольований доступ до даних, відокремлюючи внутрішню реалізацію від зовнішнього інтерфейсу.

Сеттери (Mutators)

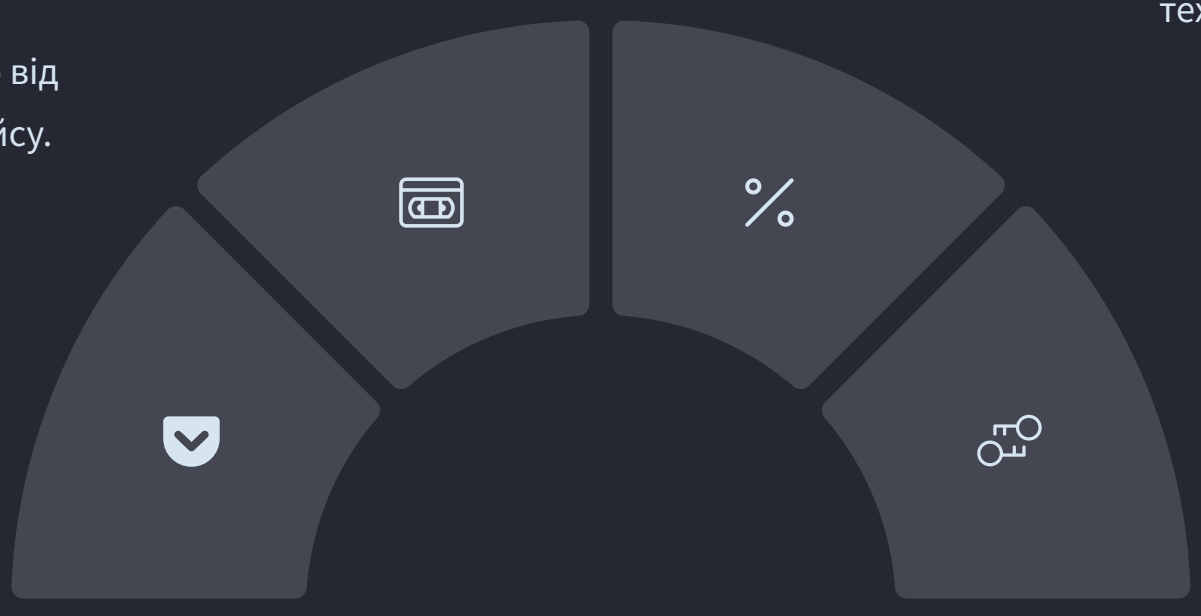
Методи, що дозволяють змінювати значення приватних полів. Вони можуть включати валідацію вхідних даних, забезпечуючи цілісність стану об'єкта.

Властивості (Properties)

У деяких мовах (C#, Python) є спеціальні синтаксичні конструкції, що поєднують геттери і сеттери в єдиний концепт властивості.

Приватні поля

Оголошення полів класу як приватних — базова техніка інкапсуляції, що запобігає прямому доступу ззовні.



Приклади інкапсуляції у різних мовах програмування:

Java

```
public class BankAccount {
    // Приватні поля
    private double balance;
    private String accountNumber;

    // Конструктор
    public BankAccount(String accountNumber) {
        this.accountNumber = accountNumber;
        this.balance = 0.0;
    }

    // Геттер
    public double getBalance() {
        return balance;
    }

    // Немає прямого сеттера для balance
    // Замість цього спеціалізовані методи
    public void deposit(double amount) {
        if (amount > 0) {
            balance += amount;
        }
    }

    public boolean withdraw(double amount) {
        if (amount > 0 && balance >= amount) {
            balance -= amount;
            return true;
        }
        return false;
    }
}
```

Python

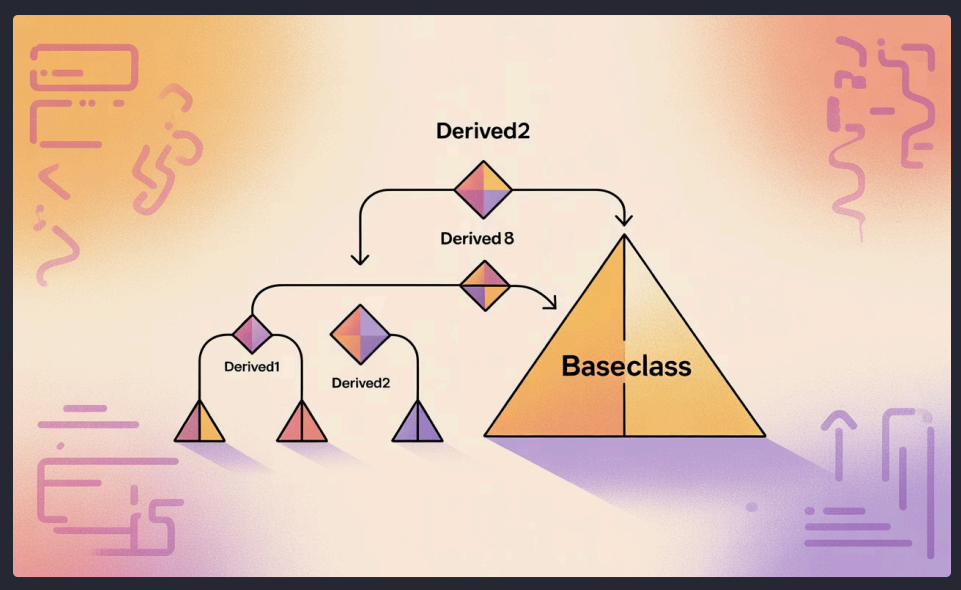
```
class BankAccount:
    def __init__(self, account_number):
        self.__account_number = account_number
        self.__balance = 0.0

    @property
    def balance(self):
        return self.__balance

    # Немає сеттера для balance

    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount

    def withdraw(self, amount):
        if amount > 0 and self.__balance >= amount:
            self.__balance -= amount
            return True
        return False
```

Як видно з прикладів, інкапсуляція дозволяє контролювати модифікацію даних та забезпечує їх цілісність. У банківському акаунті баланс не може бути змінений напряму (немає сеттера), а лише через методи депозиту та зняття коштів, які включають валідацію. Це запобігає встановленню некоректних значень і гарантує, що операції відповідають бізнес-правилам (наприклад, не можна зняти більше коштів, ніж є на рахунку).

Інкапсуляція не лише захищає дані, але й робить код більш модульним та легким для розуміння. Розробники можуть використовувати клас через його публічний інтерфейс, не замислюючись про внутрішні механізми. Крім того, інкапсуляція дозволяє змінювати внутрішню реалізацію класу без впливу на зовнішній код, що значно полегшує підтримку та розвиток програмного забезпечення.

Спадкування: повторне використання та розширення коду

Спадкування — це механізм ООП, який дозволяє одному класу (похідному або підкласу) успадковувати властивості та методи іншого класу (базового або суперкласу). Це одна з найпотужніших концепцій ООП, що забезпечує повторне використання коду та встановлення ієрархічних відносин між класами.



Основна ідея спадкування полягає в тому, що новий клас може бути створений на основі існуючого, з додаванням нових функцій або модифікацією існуючих. Це дозволяє будувати ієрархії класів, де більш загальні характеристики визначені у базових класах, а специфічні деталі — у похідних.

Одиночне спадкування (Single Inheritance)

У цій моделі кожен клас може успадковувати лише від одного базового класу. Це спрощує структуру програми та запобігає проблемам, що можуть виникнути при множинному спадкуванні. Мови Java, C#, Python підтримують одиночне спадкування для класів.

Множинне спадкування (Multiple Inheritance)

Дозволяє класу успадковувати від кількох базових класів одночасно. Хоча це надає більшу гнучкість, воно також вводить потенційні проблеми, такі як конфлікт імен (проблема ромба). Мови C++ та Python підтримують множинне спадкування, тоді як Java і C# уникають його, використовуючи інтерфейси.

Ієрархічне спадкування (Hierarchical Inheritance)

Кілька підкласів успадковують від одного базового класу. Це поширений шаблон в ООП, що дозволяє створювати спеціалізовані версії загального класу. Наприклад, класи "Автомобіль", "Мотоцикл" і "Велосипед" можуть успадковувати від класу "Транспортний засіб".

Багаторівневе спадкування (Multilevel Inheritance)

Клас успадковує від похідного класу, утворюючи ланцюжок успадкування. Наприклад, клас "SportsCar" успадковує від "Car", який, у свою чергу, успадковує від "Vehicle". Це дозволяє створювати глибокі ієрархії класів.

> Базовий клас (також відомий як суперклас або батьківський клас) — це клас, від якого інші класи успадковують властивості та методи. Він зазвичай представляє більш загальну концепцію.

> Похідний клас (також відомий як підклас або дочірній клас) — це клас, який успадковує від базового класу. Він зазвичай представляє більш специфічну концепцію і може додавати нові властивості та методи або перевизначати успадковані методи.

Спадкування є потужним інструментом для моделювання різноманітних відносин між концепціями. Воно дозволяє не лише повторно використовувати код, але й відображати природні ієрархії, які існують у реальному світі або в концептуальних моделях. Однак важливо використовувати спадкування з обережністю, дотримуючись принципу "є" (is-a) — підклас повинен дійсно бути спеціалізацією базового класу, а не просто використовувати його функціональність.

Особливості реалізації спадкування

Хоча концепція спадкування універсальна для всіх об'єктно-орієнтованих мов, її реалізація може суттєво відрізнятися. Розглянемо ключові слова та механізми, що використовуються для реалізації спадкування в популярних мовах програмування, а також типові обмеження.

		3	
Java	C++	Python	C#
Використовує ключове слово **extends** для успадкування від класу та **implements** для реалізації інтерфейсів. Не підтримує множинне спадкування для класів, але дозволяє класу реалізувати кілька інтерфейсів.	Використовує двокрапку **:** і модифікатори доступу (**public**, **protected**, **private**) для визначення типу успадкування. Повністю підтримує множинне спадкування.	Вказує базові класи в дужках після імені класу. Підтримує множинне спадкування з власним механізмом вирішення порядку методів (MRO - Method Resolution Order).	Використовує двокрапку **:** для успадкування, де перший клас після двокрапки є базовим, а решта — інтерфейсами. Не підтримує множинне спадкування для класів.

> Приклади реалізації спадкування у різних мовах:

Java

```
// Базовий клас
public class Animal {
    protected String name;

    public Animal(String name) {
        this.name = name;
    }

    public void makeSound() {
        System.out.println("Some generic animal
sound");
    }
}

// Похідний клас
public class Dog extends Animal {
    private String breed;

    public Dog(String name, String breed) {
        super(name); // Виклик конструктора
базового класу
        this.breed = breed;
    }

    // Перевизначення методу базового класу
    @Override
    public void makeSound() {
        System.out.println("Woof! Woof!");
    }

    // Новий метод, специфічний для собак
    public void fetch() {
        System.out.println(name + " is fetching the
ball!");
    }
}
```

Python

```
# Базовий клас
class Animal:
    def __init__(self, name):
        self.name = name

    def make_sound(self):
        print("Some generic animal sound")

# Приклад множинного спадкування
class Mammal:
    def give_birth(self):
        print("Giving birth to live young")

# Похідний клас з множинним спадкуванням
class Dog(Animal, Mammal):
    def __init__(self, name, breed):
        # Виклик конструктора базового класу
        super().__init__(name)
        self.breed = breed

    # Перевизначення методу базового класу
    def make_sound(self):
        print("Woof! Woof!")

    # Новий метод, специфічний для собак
    def fetch(self):
        print(f"{self.name} is fetching the ball!")
```

Спадкування, хоча й потужний механізм, має свої обмеження та потенційні проблеми, які розробники повинні враховувати:

- **Проблема ромба (Diamond Problem)** — виникає при множинному спадкуванні, коли клас успадковує від двох класів, які мають спільного предка. Це може призвести до неоднозначності щодо того, яку версію методу використовувати.
- **Крихкість базового класу** — зміни в базовому класі можуть неочікувано вплинути на похідні класи, порушуючи їх функціональність.
- **Тісне зв'язування** — спадкування створює сильний зв'язок між базовим і похідним класами, що може ускладнити розвиток системи.
- **Неправильне використання** — не всі відносини між класами повинні моделюватися через спадкування. Іноді композиція (класс має об'єкт іншого класу) є більш підходящим рішенням.

Розуміння цих особливостей та обмежень спадкування допомагає розробникам ефективно використовувати цей механізм, уникаючи типових помилок проектування.

Поліморфізм: багато форм для гнучкості

Поліморфізм (від грецьких слів "полі" - багато і "морф" - форма) — це один з фундаментальних принципів ООП, який дозволяє використовувати об'єкти різних класів через єдиний інтерфейс. Суть поліморфізму полягає в тому, що один і той самий метод може мати різну реалізацію в різних класах, забезпечуючи різну поведінку для різних типів об'єктів.

Поліморфізм тісно пов'язаний зі спадкуванням, оскільки саме механізм спадкування дозволяє створювати ієрархії класів, де похідні класи можуть перевизначати методи базових класів, забезпечуючи свою власну реалізацію.

У об'єктно-орієнтованому програмуванні виділяють два основні типи поліморфізму:

Статичний поліморфізм (Compile-time)

Також відомий як раннє зв'язування. Визначається під час компіляції програми. Основний механізм статичного поліморфізму — перевантаження методів.

- **Перевантаження методів (Method Overloading)**: Визначення кількох методів з однаковим ім'ям, але різними параметрами (кількість, тип, порядок) в одному класі. Компілятор вибирає правильний метод на основі аргументів при виклику.

Динамічний поліморфізм (Run-time)

Також відомий як пізнє зв'язування. Визначається під час виконання програми. Основний механізм динамічного поліморфізму — перевизначення методів.

- **Перевизначення методів (Method Overriding)**: Визначення методу в похідному класі з тим самим ім'ям, типом повернення та параметрами, що й метод базового класу. Під час виконання програми система визначає, яку версію методу викликати, на основі фактичного типу об'єкта.

> Приклад перевантаження методів (статичний поліморфізм):

```
public class Calculator {
    // Перевантажені методи add з різними параметрами
    public int add(int a, int b) {
        return a + b;
    }

    public double add(double a, double b) {
        return a + b;
    }

    public int add(int a, int b, int c) {
        return a + b + c;
    }
}

// Використання
Calculator calc = new Calculator();
int sum1 = calc.add(5, 10);          // Викликає першу версію
double sum2 = calc.add(5.5, 10.5);   // Викликає другу версію
int sum3 = calc.add(5, 10, 15);      // Викликає третю версію
```

> Приклад перевизначення методів (динамічний поліморфізм):

```
// Базовий клас
public class Shape {
    public void draw() {
        System.out.println("Drawing a shape");
    }
}

// Похідні класи
public class Circle extends Shape {
    @Override
    public void draw() {
        System.out.println("Drawing a circle");
    }
}

public class Rectangle extends Shape {
    @Override
    public void draw() {
        System.out.println("Drawing a rectangle");
    }
}

// Використання
Shape shape1 = new Circle();
Shape shape2 = new Rectangle();

shape1.draw();  // Виведе: "Drawing a circle"
shape2.draw();  // Виведе: "Drawing a rectangle"
```

У наведеному прикладі динамічного поліморфізму, незважаючи на те, що змінні shape1 і shape2 оголошені як тип Shape, фактично вони містять об'єкти класів Circle і Rectangle. Завдяки поліморфізму, при виклику методу draw() JVM визначає фактичний тип об'єкта і викликає відповідну версію методу.

Поліморфізм є потужним інструментом для створення гнучкого та розширюваного коду. Він дозволяє працювати з об'єктами різних класів через спільний інтерфейс, що значно спрощує додавання нових типів без зміни існуючого коду. Це ключовий принцип для реалізації патернів проектування, таких як Factory, Strategy та Template Method.

10 тестових завдань з поясненнями

1 Яка з наступних концепцій ООП дозволяє об'єктам різних класів виконувати однакові дії по-різному?

A. Абстракція

B. Інкапсуляція

C. Спадкування

D. Поліморфізм

Правильна відповідь: D. Поліморфізм

Пояснення: Поліморфізм дозволяє об'єктам різних класів реагувати по-різному на один і той самий метод. Це досягається через перевизначення методів у класах спадкоємцях, що дає можливість використовувати об'єкти різних типів через спільний інтерфейс.

2 Яке ключове слово використовується в Java для успадкування від класу?

A. implements

B. extends

C. inherits

D. class

Правильна відповідь: B. extends

Пояснення: У Java ключове слово extends використовується для створення підкласу, який успадковується від іншого класу. Implements використовується для реалізації інтерфейсів, а не для успадкування від класів.

3 Що з наступного НЕ є перевагою інкапсуляції?

A. Захист даних від зовнішнього доступу

B. Контроль над станом об'єкта

C. Можливість змінювати внутрішню реалізацію без впливу на зовнішній код

D. Можливість створення ієрархії класів

Правильна відповідь: D. Можливість створення ієрархії класів

Пояснення: Створення ієрархії класів є перевагою спадкування, а не інкапсуляції. Інкапсуляція зосереджена на приховуванні внутрішньої реалізації та захисті даних, а не на встановленні відносин між класами.

4 Яка концепція ООП дозволяє розробникам зосередитися на важливих аспектах об'єкта, ігноруючи несуттєві деталі?

A. Абстракція

B. Інкапсуляція

C. Спадкування

D. Поліморфізм

Правильна відповідь: A. Абстракція

Пояснення: Абстракція дозволяє моделювати складні системи шляхом виділення найважливіших аспектів і ігнорування несуттєвих деталей. Це дозволяє розробникам створювати спрощені моделі складної реальності.

5 Який тип поліморфізму визначається під час компіляції програми?

A. Динамічний поліморфізм

B. Статичний поліморфізм

C. Пізнє зв'язування

D. Перевизначення методів

Правильна відповідь: B. Статичний поліморфізм

Пояснення: Статичний поліморфізм (також відомий як раннє зв'язування) визначається на етапі компіляції програми. Основний механізм статичного поліморфізму — перевантаження методів, коли один і той самий метод має різні сигнатури в межах одного класу.

6 Яка концепція ООП дозволяє одному класу успадковувати властивості та методи іншого класу?

A. Абстракція

B. Інкапсуляція

C. Спадкування

D. Поліморфізм

Правильна відповідь: C. Спадкування

Пояснення: Спадкування — це механізм, який дозволяє одному класу (похідному) успадковувати властивості та методи іншого класу (базового), що забезпечує повторне використання коду та встановлення ієрархічних відносин між класами.

7 Який модифікатор доступу в Java дозволяє доступ до членів класу лише всередині самого класу?

A. public

B. protected

C. private

D. default

Правильна відповідь: C. private

Пояснення: Модифікатор private в Java обмежує доступ до членів класу лише в межах самого класу. Це найсильніший рівень інкапсуляції, який повністю приховує внутрішній стан об'єкта від зовнішнього коду.

8 Який механізм динамічного поліморфізму передбачає створення методу в похідному класі з тим самим ім'ям і сигнатурою, що й метод базового класу?

A. Перевантаження методів (Overloading)

B. Перевизначення методів (Overriding)

C. Перекриття методів (Hiding)

D. Перенаправлення методів (Redirecting)

Правильна відповідь: B. Перевизначення методів (Overriding)

Пояснення: Перевизначення методів (Method Overriding) — це механізм динамічного поліморфізму, який дозволяє похідному класу надати свою власну реалізацію методу, який вже визначений у базовому класі. При цьому ім'я, тип повернення та параметри методу залишаються тими самими.

9 Яка з наступних мов програмування НЕ підтримує множинне спадкування класів?

A. C++

B. Python

C. Java

D. Ruby

Правильна відповідь: C. Java

Пояснення: Java не підтримує множинне спадкування для класів, дозволяючи класу успадковуватися лише від одного базового класу. Замість цього Java використовує інтерфейси для досягнення деяких переваг множинного спадкування без його недоліків, таких як проблема ромба.

10 Яка наступна пара "метод-концепція ООП" є неправильною?

A. Абстрактні класи - Абстракція

B. Приватні поля - Інкапсуляція

C. Extends ключове слово - Спадкування

D. Перевантаження методів - Спадкування

Правильна відповідь: D. Перевантаження методів - Спадкування

Пояснення: Перевантаження методів (Method Overloading) пов'язане з поліморфізмом, а не зі спадкуванням. Це механізм статичного поліморфізму, який дозволяє визначати кілька методів з однаковим ім'ям, але різними параметрами в одному класі. Спадкування ж стосується відносин між класами, а не перевантаження методів.